Software Hazard and Safety Analysis

John McDermid

University of York,
Heslington,
York, YO10 5DD
UK

Abstract. Safety is a system property and software, of itself, cannot be safe or unsafe. However software has a major influence on safety in many modern systems, e.g. aircraft and engine controls, railway signalling, and medical equipment.
The paper outlines the principles of system hazard and safety analysis, and briefly describes work on adapting classical hazard and safety analysis techniques to apply to software. It then briefly discusses the role of formal analysis in software hazard and safety assessment, indicating both the state of practice and the aims of some ongoing research projects.
Note: this paper is provided to support a tutorial on software hazard and safety analysis, and is not intended to be a definitive treatment of the issues.

1 Introduction

Safety is concerned with protection of human life, the environment and property. There is no such thing as absolute safety – all human endeavour has attendant risks. However we say that a system is safe (enough) if the risk of causing damage to life, the environment or property is acceptable. Normally we measure risk as a combination of the probability of damage occurring and the extent of damage, e.g. the number of lives which are expected to be lost. Acceptability of such risks is a complex issue; ultimately it is a societal judgement. Discussion of acceptability of risk is outside the scope of this paper.

Damage, as defined above, can arise in two basic ways – by uncontrolled or unintended transfer of energy, and through failure to contain harmful materials, e.g. toxins or radioactive sources (the purist might argue that this too is a failure of energy containment). Software is not a harmful substance, nor does it have high energy levels – thus it cannot be safe or unsafe of itself. However it is used in many systems where it contributes to safety, e.g. through control over hazardous physical processes [1]. We use the term *software hazard and safety analysis* to refer to the process of assessing the contribution of software to safety in its broader system context.

In essence there are four safety-relevant parts of a system development process:

- Identifying hazards and associated safety requirements;
- Designing the system to meet its safety requirements;
- Analysing the system to show that it meets its safety requirements;
- Demonstrating the safety of the system by producing a safety case.

We briefly give an overview of a typical safety process, then consider these process elements in turn.

2 Safety Processes and Software

Safety processes parallel and complement system development processes, being concerned with identifying and controlling ways in which systems may behave, or fail, so as to be unsafe. The processes are normally structured around the notion of a *hazard* – a circumstance which can lead to damage, e.g. the loss of braking on a car is a hazard. The early phases of the safety process are concerned with identifying hazards, and then determining the associated risk. If the risk is deemed unacceptable, then remedial design work must be undertaken – to make the hazard less likely to occur or to mitigate the consequences. The results of such analysis are often referred to as *derived safety requirements*, or DSRs.

Once hazard analysis has been undertaken, design and implementation can continue, with the aim of producing a system which meets all its requirements, including the DSRs. Whilst the above implies that there is a clean break between requirements and design, in practice there is a progressive shift in focus and often significant design iteration. Also there will inevitably be the need for trade-offs between different requirements, including DSRs, to meet overall project goals.

Once the design and implementation is complete, and as it is being integrated, analysis and testing is undertaken to show that the system meets its requirements, including DSRs. The analysis and test results provide evidence that the system is safe. However it is becoming increasingly common to provide a *safety case*, not just analysis results, for inspection by a third party, e.g. a certification agency. The safety case complements the evidence by providing the *arguments* which show why the evidence is (deemed) sufficient to demonstrate safety of the system.

In practice, processes are much more complex than implied above, but the essence of the activities is as described. Also, most real-world projects are governed by standards – of which there are many. References 2 to 6 are some of the more widely used standards in the defence and aerospace sector. Hermann [7] gives an overview of these, and many other, standards.

Most safety standards are concerned with how engineered systems can fail and can give rise to hazards. Software can only "fail" due to *systematic* causes, e.g. requirements or design errors, and the usual analysis processes do not apply. Instead, the community have taken the view that the best way to approach the issue is to have design processes which reduce the likelihood of introducing such flaws into software. Consequently many standards for the development of safety critical software have been written. References 8 to 10 are some of the better known standards in this area. However some standards cover both system and software issues: IEC 61508 [10] deals with system issues and the Australian Standard Def(Aust) 5679 [4] also addresses software issues. Generally these standards try to control software-related risk through the notion of *safety integrity levels*, or SILs.

Superficially these standards are very different but, on closer inspection, there are many areas of commonality, e.g. the concept of hazard, the intent to reduce risk, and so on. An attempt has been made to rationalise these different processes [11] and to show that there is significant commonality in these standards. However there is divergence in how they treat systematic issues, and this has led to a number of authors questioning the notion of SILs and the soundness of the guidance in the standards [12, 13]. We defer any further discussion of SILs to our treatment of safety cases.

3 Software Hazard Analysis and Derived Requirements

In considering software it is reasonable to assume that the early stages of the system safety process have identified hazards, and have determined those hazards to which software can contribute. We thus assume that the system safety process has identified *hazardous failure conditions* (HFCs) for the software which are conditions which are either sufficient in themselves, or sufficient in conjunction with other credible conditions, to give rise to a (system level) hazard.

Assuming also that there is a specification for the software in a system, we have two questions to answer:

- If the software functions as specified, does this give rise to any HFCs?
- Are there plausible failure modes of the software, or of the underlying computing hardware which are not contained by the software, which can give rise to HFCs?

Software hazard analysis amounts to answering these questions and determining consequential actions.

If specifications are informal then these questions can only be addressed through the use of human skill and judgement. The majority of techniques used at this stage of the process are informal, based on functional models of the system – the method perhaps most widely used in the aerospace sector is *functional failure analysis* (FFA) (see [6]). FFA uses three “guidewords” to prompt analysis:

- Function provided when not intended (commission);
- Function not provided when required (omission)
- Function provided incorrectly.

The analysis considers each function in turn and decides whether or not these *hypothetical* failure modes are credible and, if they are, what the consequences might be. For failure modes which are deemed credible, and where the consequences are severe, some remedial action will be identified. The remedial action may be for a specification change if the problem is “deep seated”, or to produce

DSRs for system components, e.g. to detect and mitigate failures, perhaps by using alternative sensors or control algorithms.

This type of analysis is judgemental and error prone. Indeed it has been estimated that typical analyses only identify 80% of hazards (or high level causes of hazards). It is thus tempting to consider the use of formal methods so that the analysis can be made more certain.

If the specifications are in the form of state machines, and the HFCs can be formalised, then the first question above amounts to model checking. This is not the place for a survey of model-checking techniques, but references 14–16 give an idea of the state of the art in this area. Where specifications are not in the form of state machines it is less clear cut how to address the first question, see the discussion of research issues below.

Considering the second question amounts to applying the FFA guidewords, or a more precise interpretation thereof, to the specification to produce a "mutated" specification incorporating possible failure scenarios. Next we need to determine whether or not any HFCs can arise from the "mutated" specification. Typically we have transitions annotated with labels of the form:

$$e[c]/a$$

to be read "when event e occurs and condition c holds then (take the transition and) perform action a". Omission can be characterised as "a does not happen when event e occurs and condition c holds", and so on. The third guideword needs the most interpretation, as the notion of erroneous can be taken to mean inappropriate transitions taken, inappropriate actions, and so on. This gives rise to problems of combinatorial explosion, see the research issues below.

Analysis of the "transition mutations" shows that they reduce to a relatively small number of regular forms, all of which can be represented as additional transitions in the state machine (inevitably introducing non-determinism). Once more model checking can be used to see if HFCs can arise – although the computational cost is much higher. In fact it is possible to automate the generation of the mutated state machines, and then to determine which mutants, if any, give rise to the HFCs (Reference 17 describes the overall approach in the context of UML, although not the details of the automation.)

The results of this "hazard analysis" need some interpretation. First, some situations which are formally possible may be physically impossible, and must be discounted. Second, some may show deep flaws in the system concept, and thus must give rise to a change in the system specification. Third, the analysis may identify DSRs on parts of the design which are critical to avoiding the HFCs. Generally these will be simpler than full functional correctness, e.g. correct operation of an interlock, and will be the focus of more detailed design and analysis. These DSRs must be represented in a means which is compatible with the specification approach used; in our case we have chosen to use a form of rely-guarantee conditions [17].

The approach outlined above is intended to be generic, although we have appealed to work in York for more concrete illustrations of the ideas. Similar ideas can be found elsewhere, e.g. in Leveson's SpecTRM method [18].

4 Software Design and Implementation

There are two important aspects to software design and implementation – the software engineering process and the (software) safety process. Many academics advocate the use of refinement from (formal) specifications, and it might be thought that safety is an area where such processes ought to be used (indeed this is one of the requirements of DS 00-55). However using refinement is not so straightforward, as we now endeavour to explain.

The idea of program refinement [19] goes back over 30 years, and the idea has been extended into a formal framework, e.g. the seminal work of Carroll Morgan [20]. However it has become acknowledged that refinement has its difficulties and, for example, non-functional properties such as safety and security are not necessarily preserved through refinement, e.g. weakening a pre-condition may admit an unsafe behaviour which is not present in the more abstract specification. Problems with refinement have been known for some time [21]. More recently, acknowledgement of these problems has led to the introduction of the notion of retrenchment [22] which tries to find formally defensible ways of developing programs whilst breaking the standard rules of formal refinement. However we do not view retrenchment, in its current state, as being mature enough to apply to real systems. Instead we assume a more informal approach to developing programs has to be adopted, and thus turn our attention to the safety process.

The safety process has two main concerns:

– Flowing down DSRs to low level components;
– Assessing the design for additional potential contributions to HFCs, and hence deriving further DSRs.

The first of these is part of the normal "requirements flow down" in system development, except that our concern is only with DSRs. In system safety a more systematic approach is used. This is based on fault trees, and known as *preliminary system safety assessment* (PSSA) [6]. In the software case, the fault tree would be built from an HFC down to the level of failure modes of software components. The component level DSRs are (safety) properties which the component must guarantee for the system to behave safely. Unfortunately, the guidance in the system safety standards is inadequate for dealing with software based systems [23], and considerable judgement is needed for this part of the process. Perhaps the best description of the approach is in Leveson's SpecTRM [18].

The second safety concern requires us to consider ways in which any "extra" functionality introduced in producing the design may contribute to HFCs. In a way the problem is like that of hazard analysis except that, at this stage, it is possible to say much more about which types of failure mode are credible as much more is known about the design and implementation, e.g. the mapping of software to the computing hardware. Approaches like FFA are sometimes employed, but it is more common to use adaptations of HAZOP as this considers both causes and consequences of deviations from intended behaviour (Leveson uses the term "deviation analysis").

HAZOP is similar to FFA, in that it hypothesises failure modes, but the range of guidewords is much greater, including: early, late, too much, too little, etc. Adaptations of HAZOP to computer systems have sometimes simply accepted the normal HAZOP guidewords [24] (originally developed for the chemical industry) or sought to adapt them to computer systems and software [25]. In principle model checking could be applied in the same way as described at specification level – but too our knowledge this work has only ever been done manually, due to the complexity of the designs which need to be analysed.

In theory, new hazardous behaviours can be introduced at each level in the design decomposition, thus design analysis needs to be repeated at each level. Such analysis could be done, but it would be very onerous. Our experience is that the analysis tends to be done once at a level quite close to the code – where there is a simple refinement to the implementation. In other words, it is carried out a level where no more new potentially hazardous behaviours will be introduced in the development process.

Ultimately software has to be realised by implementation in a programming language, and the DSRs "flowed down" to the program – in the form of pre- and post-conditions. There is really only one commercial tool which supports such a use of formal annotations and formal analysis – the SPARK Examiner [26]. We discuss program level issues in more detail in the next section, but first consider some trends in software development which have an impact on the way software is developed and analysed.

The discussion above implicitly assumes that al programs are produced manually from a specification or, more likely, at the "bottom" of a hierarchy of specifications. This is probably true of the majority of current safety critical applications, but there are trends to the use of greater automation.

Current projects are considering, or using, design notations such as Matlab/ Simulink [27] and employing such tools to generate code from the designs. This is somewhat in conflict with the use of classical formal verification approaches, as these tools do not usually include pre- and post-conditions in the code. In theory, if the code generators were trustworthy, this would not be an issue. However, these tools are continually evolving so it is difficult to be confident in their output – without further checks. This is perhaps mainly a research issue at present, but it will become more of an issue as code generation is more routinely used for generating critical code. This suggests the need for analysis techniques which are independent of the way in which the code is produced, and which do not require insertion of pre- and post-conditions in the code.

5 Software Safety Analysis

The aim of software safety analysis is to show that the software meets the DSRs, locally, and overall does not contribute (in unacceptable ways) to the HFCs. In principle this can be done using safety analysis techniques or using software engineering techniques. We believe that it is most appropriate to employ software engineering methods, but we start by considering safety analysis techniques as this gives a basis for explaining this point of view.

Researchers in software safety have adapted standard system safety techniques, e.g. fault trees and failure modes and effects analyses to software, but with mixed success. Perhaps the most fully developed approach is due to Leveson [28, 29] who pioneered the application of fault trees to software. In essence software fault-tree analysis is a modified form of wp-calculus, but focusing on causes of HFCs or violations of DSRs, not establishing partial correctness. Our experience, and that of others, is that software fault trees work well in particular circumstances, but are difficult to apply to large programs, without mechanical support for expression evaluation, etc. There is a growing view that the use of static code analysis and proof techniques is much more cost-effective, especially when tool supported. (we consider testing below.)

In the UK, Defence Standard 00-55 [9] requires the application of static code analysis to safety-critical software. The standard has probably not been applied in its entirety, but elements of it have been used on a number of projects. For example, static code analysis techniques have been applied retrospectively to the software on the C130J aircraft. All the code was developed to the requirements of DO178B. Some of the code was written in SPARK Ada [26], and hence used the SPARK tools. Other code in C, full Ada, etc. was analysed using Malpas [30]. The approach was initially to apply static analysis in a blanket way, but later this was refined to use a hazard directed approach, i.e. focusing on DSRs and HFCs.

There are limited publications on the work but there are some “snapshots” of the project, e.g. reference 31. To the author’s knowledge about 550 kLoC of code has been analysed. Initially around 50 potentially safety critical code anomalies were found, but these were removed in later builds of the software (and the error correction suitably verified). Interestingly, the project found that informal familiarisation with code was by far the most effective and cost-effective way of finding faults – followed by full semantic analysis (i.e. proof). Simpler and cheap static analysis, e.g. information and data flow, found relatively few problems. This shows the capability of the technology, applied to software developed in conventional ways.

More recently, QinetiQ have been applying a rather different approach to software which has been derived from control law definitions, including Matlab models. The approach has been to use an intermediate language (Z) and to translate the specifications and code into a common form, and to verify equivalence. As the structure of the conjectures produced is very regular it is possible to use tactics to automate the analysis. The principles are discussed in reference 32, although this does not discuss practical applications in detail. At the time of writing, about 80% of a build of the EuroFighter flying control software had been analysed in this way, and QinetiQ aim to verify 100% of the next build – fully automatically. (Note: this provides much of the capability we identified at the end of section 4.) Interestingly, relatively few anomalies have been found, and there are questions about the cost-effectiveness of the approach, especially as the specifications being used as the basis for verification are very low level.

Note that this is concerned more with program-specification conformance than verification that the programs do not violate DSRs.

Other researchers have used model checking on specifications, to show that the software as specified (at a detailed level) does not contradict DSRs. This contrasts with the QinetiQ approach as it is abstracting from low-level specifications to system-level properties, not addressing specification-code correspondence. An example of the use of this approach on a model of an aircraft system is given in reference 33. Arguably this approach is more compelling than that used by QinetiQ as it is more naturally hazard directed. A potential limitation of this approach is that it does not address errors that might be introduced at program level. However, such issues are addressed by conventional verification activities – and the capacity to introduce new safety problems is very limited if there is a true refinement between the low level specification and the program.

As indicated above, although there needs to be a focus on safety, there is also a need to show that programs function as intended. There is no value in replicating such work in assessing safety. Thus we take it as read that testing is undertaken, both to show that the software behaves as expected on real hardware, and as part of the overall validation process. It is worth observing that many safety problems relate to requirements errors, not mistakes in coding. Thus testing has an important role in validating safety requirements. Indeed, it is common to do "fault injection testing" as may be difficult to validate fault detection and recovery specifications (including DSRs) any other way. There are other interesting testing issues but, given the remit of the tutorial, we have focused on the application of formal techniques to safety properties, and will not consider testing further.

So far we have focused on functional properties of programs, but we also need to consider non-functional issues, e.g. timing. With some classes of processor, timing properties are amenable to automated (formal) analysis [34]. However modern processors pose significant challenges for static timing analysis [35], and a combination of static analysis and testing must be used. Space precludes further discussion of non-functional properties of programs.

In general, the choice of suitable combinations of analysis techniques is a complex issue. Many of the software safety standards identify suitable (recommended) sets of techniques for developing and assessing software, although there is surprising divergence between the recommendations of different standards [11]. We briefly return to this point when discussing the software safety case.

6 Software Safety Cases

The notion of a safety case was introduced by Lord Cullen following the Piper Alpha disaster [36]. In essence a safety case consists of arguments why a system is believed to be safe enough to be operated. This argument is backed up by supporting evidence, e.g. test and other analysis results. In many situations a safety case report will be produced which contains the primary arguments, with the supporting evidence relegated to other documents or electronic media, due to their bulk.

In current industrial practice software safety is usually argued by appeal to a process, i.e. safety is asserted to arise from use of a process appropriate for a given SIL. For example, in DO178B, an accomplishment summary is produced that shows that key parts of the process have been followed; the requirements, e.g. for independent checks on activities, grow more stringent with the severity of the HFCs. There are growing doubts about the validity of the approach, for example the C130J analysis showed no noticeable correlation between SIL (DAL in DO178B terminology) and fault density. Some other concerns about use of SILs are set out in reference 13. One of the key concerns is that DO178B and other standards seem more to be governing quality, than safety, in that there is little focus on HFCs.

An alternative form of safety case would provide evidence that the software meets relevant DSRs and makes only acceptable contributions to HFCs. It is intended that the adaptation of classical safety analysis techniques to software, and the use of formal analysis on specifications/programs, can facilitate the production of such a safety case. However this is far from current practice. Further, although there is some informal acceptance of this approach to software safety cases the community is not in agreement as to what constitutes adequate software safety evidence. For example, how, if at all, should the software safety evidence vary with the criticality of the hazard? Some attempt is being made to develop a systematic approach to software safety arguments and evidence, see for example [36], but much remains to be done. This leads naturally to the identification of some research issues – to which we now turn.

7 Research Issues

There are research issues to be addressed at all stages in the safety process, not just for safety cases. Some of the more important issues are:

- How can formal approaches be used to explore the potentially hazardous failure behaviour of systems specified using techniques other than state charts? What are suitable requirements representations, and how can the hypothetical failure modes (omission, commission, etc.) be formalised and the analysis automated?
- What are appropriate HAZOP/FFA guidewords to apply to software designs? The work in reference 25 suggests one approach, but there are other possibilities and, by its very nature, a set of failure guidewords is hard to validate.
- How should non-functional properties of specifications be analysed – at all levels in the design process? Can formal analysis be extended to address application domain properties such as stability of control systems [38]?
- How can changes in designs and specifications be assessed efficiently?
- How can model-checking and other techniques be enhanced to deal with the challenges of "combinatorial explosion" that arise when considering failure behaviours as well as "intended" behaviour?

- What constitutes “sufficient” software safety evidence, and how should this vary with severity of hazard, acceptable probability of occurrence of the HFC, and so on?
- How can hazard and safety analysis techniques be applied to modern software engineering approaches such as object-oriented design? How can domain-specific tools such as Matlab be employed most effectively in a system safety process?
- What is an appropriate balance between automation and human involvement in the safety process [39]? Automation is essential to deal with problems of scale – but it is necessarily based on models, not reality. Humans are good at extrapolating beyond models; automata are not. How do we get the best of human and machine capabilities?

Addressing these research issues requires a combination of skills in software engineering, safety engineering, theory of computer science (especially in the area of optimising model checking, without making it unsound), tool development, and perhaps in certain application domains. Many of the challenges are also long-term, as the scale and complexity of systems and software being developed are growing faster than our ability to analyse them. This is an area where co-operation between diverse research groups is needed to make significant progress.

8 Conclusions

Software safety is an immature discipline – yet it is an important one due to the ever-growing reliance of modern, complex, systems on computers and software to function safely. For many years, hazard and safety analysis have been carried out informally and, despite the strictures of standards such as 00-55 in the UK, most practical software safety assessment has relied on testing and review. However things are now changing.

The capability of formal techniques, and the capacity of modern computers, means that it is becoming increasingly practical to apply automated analyses to realistic systems, as the C130J and EuroFighter examples show. However there remain many research challenges and it is clear that automation is not a panacea. One of the biggest issues that needs to be addressed is how to use automation to deal with problems of scale, whilst enabling human judgement to be applied at critical points in the process.

Acknowledgements

This work is funded in part by the Engineering and Physical Science Research Council (EPSRC) under grant GR/R70590/01.

References

1. Leveson, N.G., *Safeware: System Safety and Computers*, Addison Wesley, 1995.
2. US Department of Defense, *Military Standard 882C (Change Notice 1): System Safety Program Requirements*, 1996.
3. UK Ministry of Defence, *Defence Standard 00-56 Issue 2: Safety Management Requirements for Defence Systems*, 1996.
4. Australian Department of Defence, *Australian Defence Standard Def(Aust) 5679: Procurement of Computer-based Safety Critical Systems*, 1998.
5. Society of Automotive Engineers Inc, *Aerospace Recommended Practice (ARP) 4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems*, 1996.
6. Society of Automotive Engineers Inc, *Aerospace Recommended Practice (ARP) 4761: Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment*, 1996.
7. Hermann, D., *Software Safety and Reliability*, IEEE Computer Society Press, 1999.
8. RTCA and EUROCAE. *Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Commission for Aeronautics RTCA DO-17B/EUROCAE ED-12B, 1993
9. UK Ministry of Defence, *Defence Standard 00-55 Requirements of Safety Related Software in Defence Equipment*, 1997
10. IEC (International Electrotechnical Commission). *IEC-61508: Functional safety of electrical/electronic/ programmable electronic safety-related systems*, 1997.
11. Y Papadopoulos, Y., McDermid, J. A., *The Potential for a Generic Approach to the Certification of Safety-Critical Systems in the Transportation Sector*, Reliability Engineering and System Safety, Vol. 63, Issue 1, 1999.
12. Redmill, F. *Safety Integrity Levels – Theory and Problems*, in Lessons in System Safety, Proceedings of the Eighth Safety-Critical Systems Symposium, Springer Verlag, 2000.
13. McDermid, J. A., *Software Safety: Where's the Evidence?*, in Proc. 6th Australian Workshop on Industrial Experience with Safety systems and Software, Australian Computer Society, 2001.
14. Clarke, E.M., Grumberg, O., Peled, D.A., *Model Checking*, The MIT Press, 1999
15. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J., *Symbolic Model Checking:* 10^{20} *States and Beyond*, Information and Computation, Volume 98, Number 2, 1992.
16. Clarke, E., Grumberg, O., Somesh, J., Lu, Y., Veith, H., *Progress on the State Explosion Problem in Model Checking*, in Informatics: 10 years Back. 10 Years Ahead, Wilhelm, R. (Ed.), LNCS 2000, Springer Verlag, 2001.
17. Hawkins R. D., McDermid, J. A., *Performing Hazard and Safety Analysis of Object Oriented Systems*, in Proceedngs of ISSC, Denver, August 2002.
18. Leveson, N. G., *Safeware Engineering Corporation – SpecTRM*, `http://www.safeware-eng.com/`.
19. Wirth, N., *Program Development by Stepwise Refinement*, Communications of the ACM, Volume 14, Number 4, 1971.
20. Morgan, C. C., *Programming from Specifications*, Prentice Hall, 1994.
21. Neilsen, D. S., *From Z to C: Illustration of a Rigorous Proof Method*, DPhil Thesis, Oxford 1989.
22. Banach, R., Poppleton, M., *Sharp Retrenchment, Modulated Refinement, and Simulation*, Formal Aspects of Computing, 11, 498–540, 1999

23. S K Dawkins, S. K., Kelly, T. P., McDermid, J. A., Murdoch, J., Pumfrey, D. J., *Issues in the Conduct of PSSA*, In Proceedings of ISSC, Orlando, 1999
24. UK Ministry of Defence, *Defence Standard 00-58: HAZOP Studies on Systems Containing Programmable Electronics*, 1996.
25. McDermid, J. A., Pumfrey, D. J., *A Development of Hazard Analysis to aid Software Design*, in Proceedings of COMPASS'94, Gaithersburg, 1994.
26. Barnes, J. G., *High Integrity Ada: The SPARK Approach*, Addison Wesley, 1997.
27. `http://www.mathworks.com/`
28. Leveson, N. G., Harvey, P. R., *Software Fault Tree Analysis*, Journal of Systems and Software, 1983.
29. Leveson, N. G., Shimeall, T. J., *Safety Verification of Ada Programs using Software Fault Trees*, IEEE Software, 1991.
30. `http://www.tagroup.co.uk/malpas.htm`
31. Harrison, K. J., *Static Code Analysis on the C-130J Hercules Safety Critical Software*, Aerosystems International, 1999
32. O'Halloran, C., Smith, A., *Verification of Picture-Generated Code*, in Proceedings of the 14th IEEE Conference on Automated Software Engineering, 1999
33. Damm W., et al, *Formal Verification of an Avionics Application using Abstraction and Model Checking*, in Towards System Safety, F Redmill, F., Anderson, T. (Eds), Springer Verlag, 1999
34. Eccles, M. A., *STAMP Tool Assessment*, BAe-WSC-RP-R&D-0031, BAe Warton, 1995.
35. Bate, I. J., Conmy, P. M., McDermid, J. A., *Generating Evidence for Certification of Modern Processors for use in Safety-Critical Systems*, in Proceedings of the 5th International High Assurance Systems Engineering Symposium, Albuquerque, 2000.
36. Cullen, the Hon. Lord, *The Public Enquiry into the Piper Alpha Disaster*, HMSO, ISBN 0-10-113102, 1990.
37. Weaver, R. A., McDermid, J. A., Kelly, T. P., *Software Safety Arguments: Towards a Systematic Categorisation of Evidence*, in Proceedings of ISSC, Denver, August 2002.
38. Blow, J., Buttle, D., Galloway, A. J., *Differential Proof Contexts in SPARK*, submitted for publication, 2002.
39. Galloway, A. J., McDermid, J. A., Murdoch, J. M., Pumfrey D. J ., *Automation of System Safety Analysis: Possibilities and Pitfalls*, in Proceedings of ISSC, Denver, August 2002.